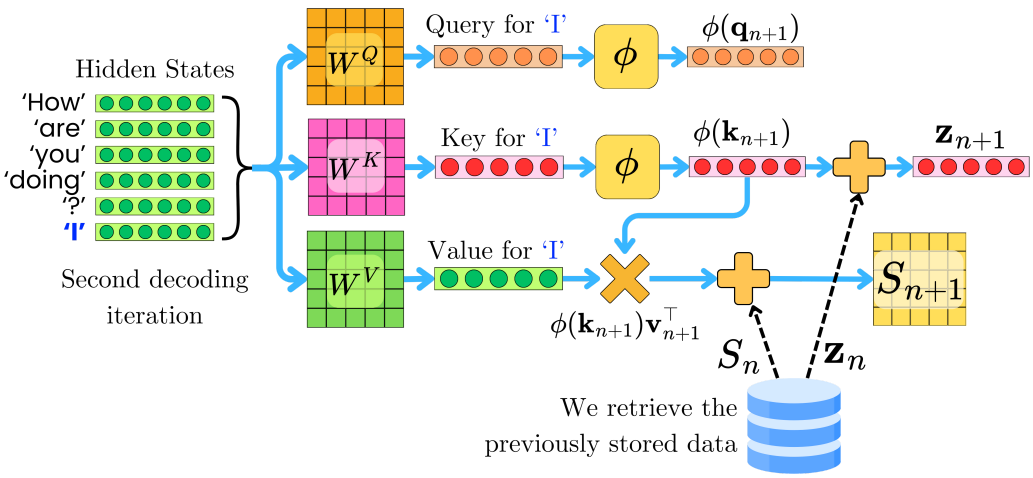
How to Linearize the Attention Mechanism

Damien Benveniste



We tend to accept the attention mechanism to be represented by the following computation:

$$C = \text{Softmax}\left(\frac{Q^\top K}{\sqrt{d_{\text{model}}}}\right) V \quad \text{or}$$

$$\mathbf{c}_i = \frac{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right) \mathbf{v}_j}{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)} \quad \text{for individual vectors} \tag{1}$$

however, this specific analytical choice is not the only one that could be chosen to fulfill the same functional role in capturing pairwise interactions between tokens. Let's review the roles of the different elements in this equation:

- **The dot-product $Q^\top K$: Similarity computation.** For each query vector, it tells you how "compatible" or similar it is to each key vector. This yields a matrix of unnormalized attention scores.
- **Normalizing by $\sqrt{d_{\text{model}}}$: Variance control.** The primary purpose of scaling by $\sqrt{d_{\text{model}}}$ is to control the scale of the attention logits before softmax, ensuring stable gradient flow and preventing the softmax from becoming too "confident" (peaked). Furthermore, extremely large logits can cause numerical instability (e.g., NaN in floating-point arithmetic), and scaling mitigates this.
- **Softmax operation: Normalization and nonlinearity.** The softmax turns the unnormalized similarity scores into a probability distribution, amplifying the effect of the most relevant keys.

- **Multiplication by V: Weighted aggregation.** Each output is a weighted sum of the values, where the weights come from the normalized similarity scores. This is how the model "mixes"information from across the input sequence

Functionally, we need a similarity function sim that is non-linear and captures the pairwise token interaction:

$$\mathbf{c}_i = \frac{\sum_{j=1}^{N} \text{sim}(\mathbf{q}_i, \mathbf{k}_j)\mathbf{v}_j}{\sum_{j=1}^{N} \text{sim}(\mathbf{q}_i, \mathbf{k}_k)} \tag{2}$$

where the denominator ensures that the similarity function is normalized to 1. If we choose $\text{sim}(\mathbf{q}_i, \mathbf{k}_j) = \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)$, we recover the softmax transformation. The *Linear Transformer* was proposed by Katharopoulos *et al* and introduced a new attention mechanism with a different analytical form, but with similar functional roles. More specifically, they suggested a similarity function where we can factorize the contribution from the keys and the queries as a product:

$$\text{sim}(\mathbf{q}_i, \mathbf{k}_j) = \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j). \tag{3}$$

In the context of kernel methods in machine learning, ϕ is called a "feature map". A feature map is a function that transforms an input vector into a new space, often a higher-dimensional one, so that a kernel function (which measures similarity) can be expressed as an inner product in that space. Essentially, ϕ extracts or "maps" the original features into a new representation where the desired similarity (that mimics the softmax behavior) is computed simply by taking a dot product. In the context of the Linear Transformer, they simply chose ϕ as follows:

$$\phi(x) = \begin{cases} x + 1 & \text{if } x > 0, \\ \exp x & \text{otherwise.} \end{cases} \tag{4}$$

This ensures that $\text{sim}(\mathbf{q}_i, \mathbf{k}_j)$ is always positive and is computationally stable. The main appeal of this linearization of the similarity kernel is the associativity property of the matrix multiplication:

$$\mathbf{c}_i = \frac{\sum_{j=1}^{N} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)\mathbf{v}_j}{\sum_{j=1}^{N} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)} = \frac{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{N} \phi(\mathbf{k}_j)\mathbf{v}_j^\top}{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{N} \phi(\mathbf{k}_j)} \tag{5}$$

For one key and one query, $\phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)$ takes d_{model} operations. Multiplying the resulting scalar alignment score to $\mathbf{v}_j$ takes another d_{model} operations. Therefore, for all the keys, computing $\sum_{j=1}^{N} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)\mathbf{v}_j$ takes $2Nd_{\text{model}}$ operations and the times complexity is $\mathcal{O}(Nd_{\text{model}})$ per query. Similarly, the denominator $\sum_{j=1}^{N} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)$ follows a time complexity of $\mathcal{O}(Nd_{\text{model}})$. Because we have N queries, the total cost is

$$\mathcal{O}(N^2 d_{\text{model}}) \quad \text{(our typical quadratic complexity!)}. \tag{6}$$

If we consider the multiplications in a different order, $\phi(\mathbf{k}_j)\mathbf{v}_j^\top$ is an outer product and results in d_{model}^2 operations. For N keys and values, we end up with Nd_{model}^2 operations for $\sum_{j=1}^{N} \phi(\mathbf{k}_j)\mathbf{v}_j^\top$. In the denominator, summing the different keys $\sum_{j=1}^{N} \phi(\mathbf{k}_j)$ requires Nd_{model} operations. Let's call $S = \sum_{j=1}^{N} \phi(\mathbf{k}_j)\mathbf{v}_j^\top$ and $\mathbf{z} = \sum_{j=1}^{N} \phi(\mathbf{k}_j)$. S is a matrix of size $d_{\text{model}} \times d_{\text{model}}$ and $\mathbf{z}$ is a vector of size d_{model}. Computing $\phi(\mathbf{q}_i)^\top S$ brings another d_{model}^2 operations,

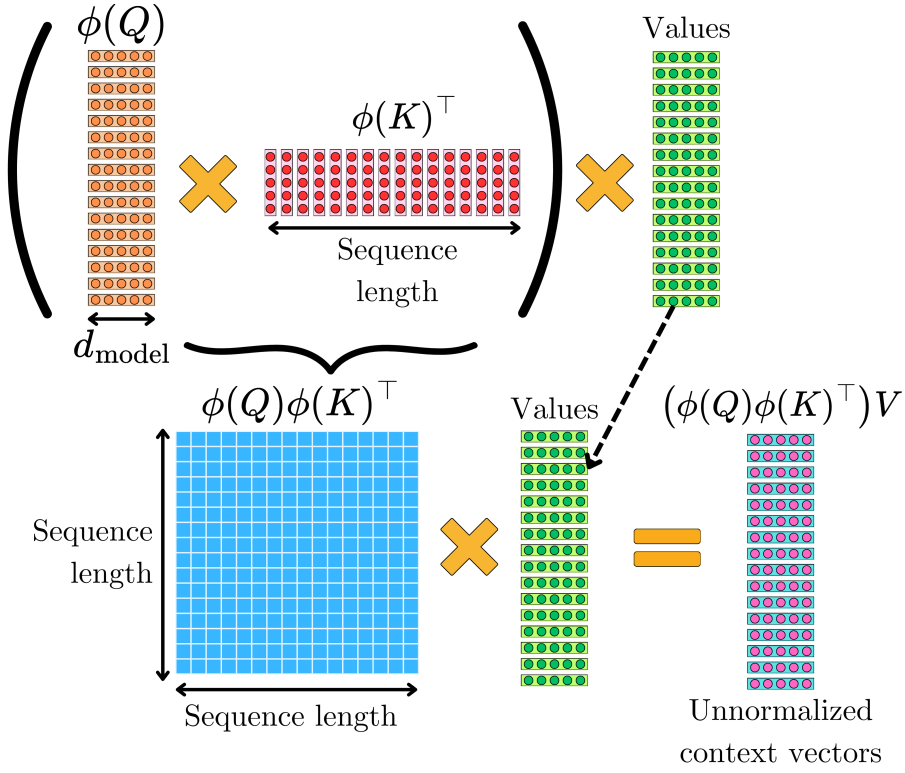


Figure 1: Vanilla attention multiplication order: quadratic complexity

and computing $\phi(\mathbf{q}_i)^\top \mathbf{z}$ takes d_{model} operations. Therefore, the cost of $\frac{\phi(\mathbf{q}_i)^\top S}{\phi(\mathbf{q}_i)^\top \mathbf{z}}$ per query is $\mathcal{O}(d_{\text{model}}^2 + d_{\text{model}}) = \mathcal{O}(d_{\text{model}}^2)$. For N queries, we obtain a total complexity of:

$$\mathcal{O}(N d_{\text{model}}^2) \quad \text{(linear complexity!)}. \tag{7}$$

By changing the order of the matrix multiplication, we were able to reduce the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$!

Decoding text with this Linear Transformer is also extremely efficient speed and memory-wise. Let's assume we have an initial prompt `"How are you doing?"`. For the vanilla attention, we first need to compute all the keys $[\mathbf{k}_1, \ldots, \mathbf{k}_n]$ and all the values $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$ but only the last query $\mathbf{q}_n$ in the sequence since we only need to predict the next token. Again, let's assume that the predict token is `'I'`, and it is appended to the input sequence `"How are you doing? I"`. In the following decoding iteration we only need the last query in the sequence $\mathbf{q}_{n+1}$, but we still need all the keys $[\mathbf{k}_1, \ldots, \mathbf{k}_n, \mathbf{k}_{n+1}]$ and all the values $[\mathbf{v}_1, \ldots, \mathbf{v}_n, \mathbf{v}_{n+1}]$. The previous keys $[\mathbf{k}_1, \ldots, \mathbf{k}_n]$ and values $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$ are exactly the same, so we could store them for the next iteration. Caching the keys and values is called KV-caching. It is computationally efficient but it requires a lot of memory to store them.

In the context of the Linear Transformer, in the first iteration, we still need to compute all

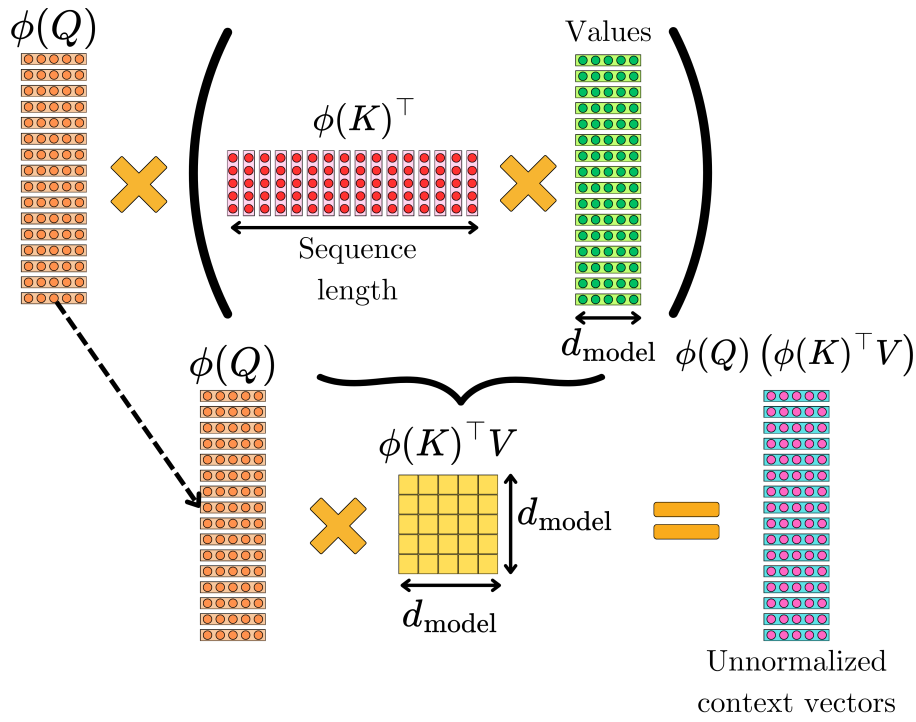


Figure 2: Different multiplication order: linear complexity

the keys $[\mathbf{k}_1, \ldots, \mathbf{k}_n]$, values $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$, and the last query $\mathbf{q}_n$ in the sequence to compute $\mathbf{c}_n$:

$$\mathbf{c}_n = \frac{\phi(\mathbf{q}_n)^\top \sum_{j=1}^{n} \phi(\mathbf{k}_j)\mathbf{v}_j^\top}{\phi(\mathbf{q}_n)^\top \sum_{j=1}^{n} \phi(\mathbf{k}_j)} \tag{8}$$

However, we can now precompute $S_n = \sum_{j=1}^{n} \phi(\mathbf{k}_j)\mathbf{v}_j^\top$ and $\mathbf{z}_n = \sum_{j=1}^{n} \phi(\mathbf{k}_j)$, and store them for the next iteration. S_n is a $d_{\text{model}} \times d_{\text{model}}$ matrix and $\mathbf{z}_n$ is a vector of size d_{model}, so a small amount of memory is necessary and it remains constant for the whole decoding process. In the following iteration, we do not need anymore the previous keys and values, and we only need to compute $\mathbf{q}_{n+1}$, $\mathbf{k}_{n+1}$, and $\mathbf{v}_{n+1}$. We can then compute the next context vector in constant time:

$$\begin{aligned} S_{n+1} &= S_n + \phi(\mathbf{k}_{n+1})\mathbf{v}_{n+1}^\top \\ \mathbf{z}_{n+1} &= \mathbf{z}_n + \phi(\mathbf{k}_{n+1}) \\ \mathbf{c}_{n+1} &= \frac{\phi(\mathbf{q}_n)^\top S_{n+1}}{\phi(\mathbf{q}_n)^\top \mathbf{z}_{n+1}} \end{aligned} \tag{9}$$

This formulation has a striking similarity to how recurrent neural networks (RNNs) operate. In an RNN, you update the hidden state $\mathbf{h}_n$ as a function of the previous state $\mathbf{h}_{n-1}$ and the current input $\mathbf{x}_n$:

$$\mathbf{h}_n = f(\mathbf{h}_{n-1}, \mathbf{x}_n) \tag{10}$$

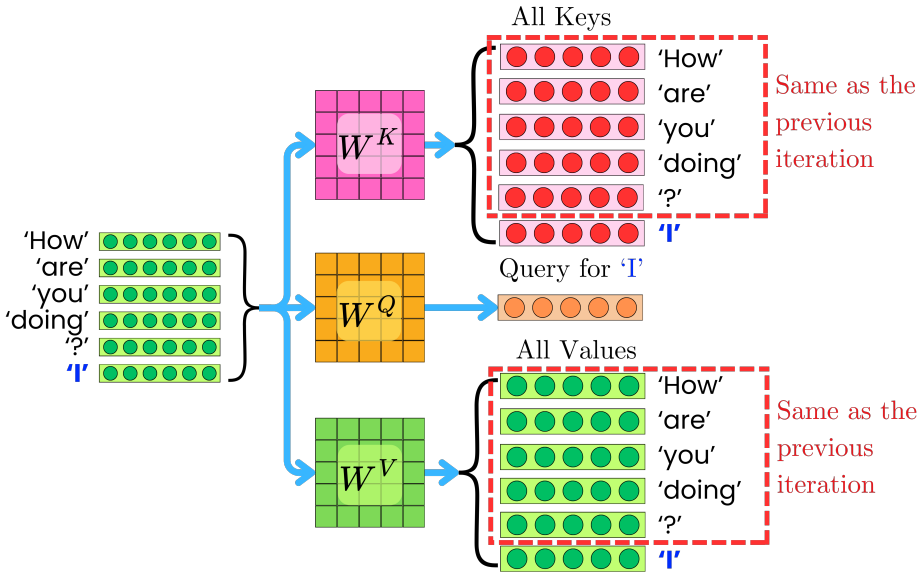


Figure 3: For the vanilla attention, the same keys and values are recomputed or stored at every iteration of the decoding process

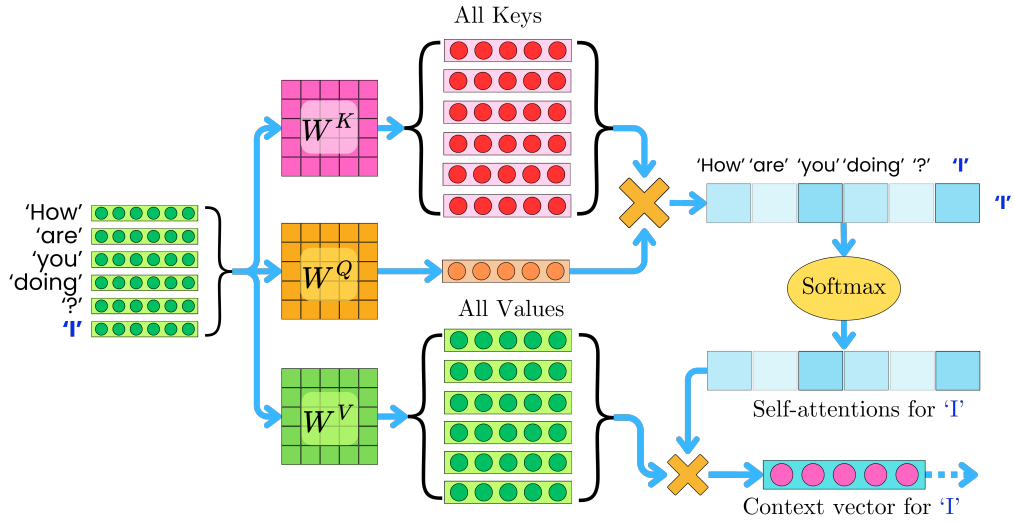


Figure 4: For the vanilla attention, we need all the keys and values to compute the next token

In the linear transformer, the cumulative sums S_n and $\mathbf{z}_n$ play the role of a hidden state that is updated with each new key–value pair.

$$(S_n, \mathbf{z}_n) = \left(S_{n-1} + \phi(\mathbf{k}_n)\mathbf{v}_n^\top, \mathbf{z}_{n-1} + \phi(\mathbf{k}_n)\right) \tag{11}$$

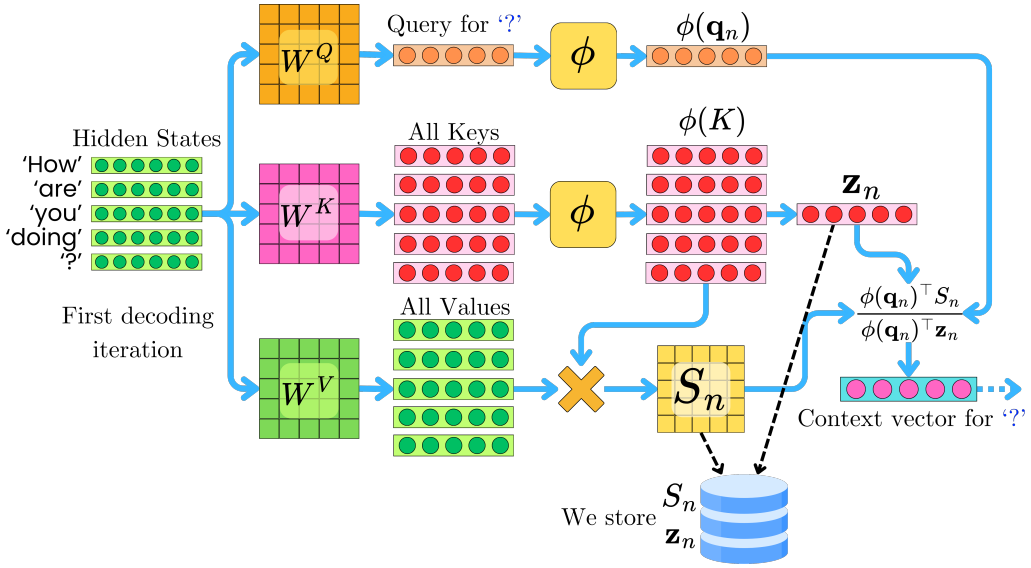


Figure 5: For the linear transformer, we first need to generate S_n and $\mathbf{z}_n$ that we store at each iteration.

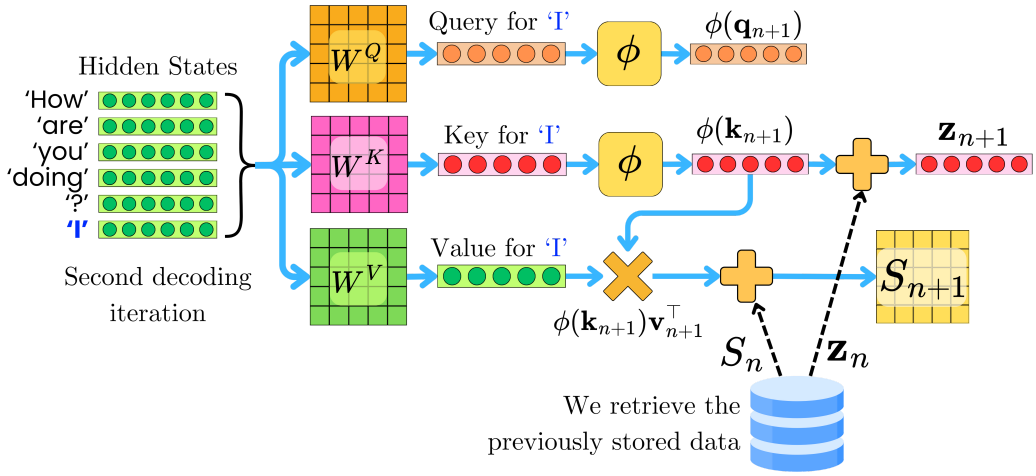


Figure 6: Because we already stored S_n and $\mathbf{z}_n$, we only need to compute $\mathbf{q}_{n+1}$, $\mathbf{k}_{n+1}$, and $\mathbf{v}_{n+1}$.

Just like an RNN's hidden state, the accumulators S_n and $\mathbf{z}_n$ have fixed sizes—independent of the sequence length N. This means that, during inference, you don't need to store all previous keys and values, you only need to maintain these fixed-size summaries. When generating a new token, you use the current query $\phi(\mathbf{k}_n)$ along with the current state $(S_n, \mathbf{z}_n)$ to compute the output $\mathbf{c}_n$.